

# ECE 601: Homework 8

Wei Xu

Email: [xu1639@purdue.edu](mailto:xu1639@purdue.edu)

Due date: 11:59 PM, Mar. 29, 2024  
(Spring 2024)

## 1 Programming Tasks

### 1.1 Generator and discriminator networks

Note: The code for this subsection is borrowed from the DLStudio package (<https://engineering.purdue.edu/kak/distDLS/>).

In this implementation, the architectures of the generator and discriminator can be found in Fig. 1, and they are defined by the class `GeneratorDCGAN` and class `DiscriminatorDCGAN` respectively. In `GeneratorDCGAN`, fake images are generated from a latent vector by upsampling with `nn.ConvTranspose2d`, while decreasing the number of channels. The first four `nn.ConvTranspose2d` layers are followed by `nn.BatchNorm2d` and `nn.ReLU`, and the last `nn.ConvTranspose2d` layer is followed by `nn.Tanh`. The tensor shape changes through the generator is  $B \times 100 \times 1 \times 1 \rightarrow B \times 512 \times 4 \times 4 \rightarrow B \times 256 \times 8 \times 8 \rightarrow B \times 128 \times 16 \times 16 \rightarrow B \times 64 \times 32 \times 32 \rightarrow B \times 3 \times 64 \times 64$ , where  $B$  is the batch size. In `DiscriminatorDCGAN`, labels are predicted by convolving the images with `nn.Conv2d`. The first `nn.Conv2d` layer is followed by `nn.LeakyReLU`, the following three `nn.Conv2d` layers are followed by `nn.BatchNorm2d` and `nn.LeakyReLU`, and the last `nn.Conv2d` layer is followed by `nn.Sigmoid` to predict the scores as labels. The tensor shape changes through the generator is  $B \times 3 \times 64 \times 64 \rightarrow B \times 64 \times 32 \times 32 \rightarrow B \times 128 \times 16 \times 16 \rightarrow B \times 256 \times 8 \times 8 \rightarrow B \times 512 \times 4 \times 4 \rightarrow B \times 1 \times 1 \times 1$ .

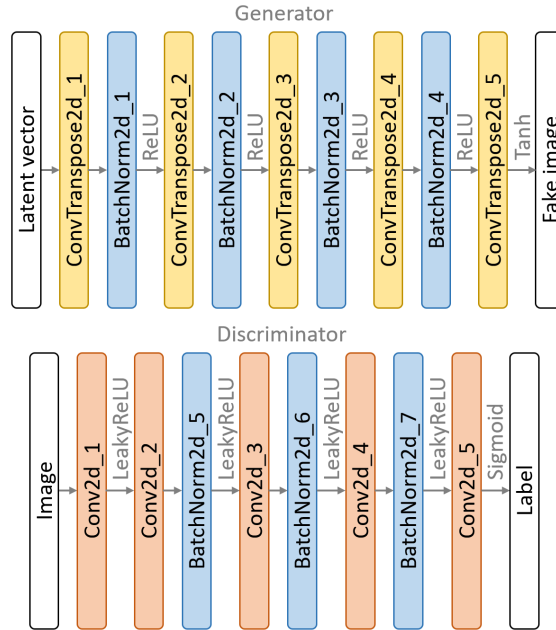


Figure 1: Architectures of `GeneratorDCGAN` and `DiscriminatorDCGAN`.

The `GeneratorDCGAN` and `DiscriminatorDCGAN` implementations are shown below. For `GeneratorDCGAN`, there are 3576704 learnable parameters and 13 layers. For `DiscriminatorDCGAN`,

there are 2766529 learnable parameters and 16 layers.

```
1 ...
2 # class for generator
3 class GeneratorDCGAN(nn.Module):
4     def __init__(self):
5         super().__init__()
6         # transposed convolutional layer with batch normalizaiton and
7         ↪ ReLU activation
8         def trans_conv_layer(in_channels, out_channels, kernel_size,
9                               ↪ stride=1, padding=0):
10             return nn.Sequential(nn.ConvTranspose2d(in_channels,
11                                                       ↪ out_channels, kernel_size=kernel_size, stride=stride,
12                                                       ↪ padding=padding, bias=False),
13                                   nn.BatchNorm2d(out_channels),
14                                   nn.ReLU(inplace=True))
15
16         # define layers
17         self.latent2image = trans_conv_layer(100, 512, 4, 1, 0)
18         self.upsampler2 = trans_conv_layer(512, 256, 4, 2, 1)
19         self.upsampler3 = trans_conv_layer(256, 128, 4, 2, 1)
20         self.upsampler4 = trans_conv_layer(128, 64, 4, 2, 1)
21         self.upsampler5 = nn.ConvTranspose2d(64, 3, kernel_size=4,
22                                               ↪ stride=2, padding=1, bias=False)
23         self.tanh = nn.Tanh()
24     def forward(self, z):
25         z = self.latent2image(z)
26         z = self.upsampler2(z)
27         z = self.upsampler3(z)
28         z = self.upsampler4(z)
29         z = self.upsampler5(z)
30         z = self.tanh(z)
31         return z
32
33 # class for discriminator
34 class DiscriminatorDCGAN(nn.Module):
35     def __init__(self):
36         super().__init__()
37         # convolutional layer with batch normalizaiton (optional) and
38         ↪ ReLU activation
39         def conv_layer(in_channels, out_channels, kernel_size, stride
40                       ↪ =1, padding=0, bn=True):
41             if bn:
42                 return nn.Sequential(nn.Conv2d(in_channels,
43                                                  ↪ out_channels, kernel_size=kernel_size, stride=
44                                                  ↪ stride, padding=padding),
45                                       nn.BatchNorm2d(out_channels),
46                                       nn.LeakyReLU(negative_slope=0.2,
47                                                     ↪ inplace=True))
48             else:
49                 return nn.Sequential(nn.Conv2d(in_channels,
50                                                  ↪ out_channels, kernel_size=kernel_size, stride=
51                                                  ↪ stride, padding=padding),
52                                       nn.LeakyReLU(negative_slope=0.2,
53                                                     ↪ inplace=True))
54
55         # define layers
```

```

41 |         self.conv1 = conv_layer(3, 64, 4, 2, 1, False) # first
    |         ↪ convolutional layer does not have batch normalization
42 |         self.conv2 = conv_layer(64, 128, 4, 2, 1)
43 |         self.conv3 = conv_layer(128, 256, 4, 2, 1)
44 |         self.conv4 = conv_layer(256, 512, 4, 2, 1)
45 |         self.conv5 = nn.Conv2d(512, 1, 4, 1, 0)
46 |         self.sig = nn.Sigmoid()
47 |     def forward(self, x):
48 |         x = self.conv1(x)
49 |         x = self.conv2(x)
50 |         x = self.conv3(x)
51 |         x = self.conv4(x)
52 |         x = self.conv5(x)
53 |         x = self.sig(x)
54 |         return x
55 | ...

```

## 1.2 Adversarial training logic

**Note:** The code for this subsection is borrowed from the DLStudio package (<https://engineering.purdue.edu/kak/distDLS/>).

For the discriminator with weights  $\theta_d$ , it is expected to output a higher score if the input image  $x$  is from the same distribution as the training data  $p_{data}$ . So the optimization problem for this part is

$$\max_{\theta_d} \mathbb{E}_{x \sim p_{data}} [\log D(x)] \quad (1)$$

For the generator with weights  $\theta_g$ , the input latent vector  $z$  is from the probability distribution  $p_Z$ . Its goal is to generate a fake image but let the discriminator give a high score. So the optimization problem for this part is

$$\min_{\theta_g} \mathbb{E}_{z \sim p_Z} [\log(1 - D(G(z)))] \quad (2)$$

These two above can be combined as

$$\min_{\theta_g} \max_{\theta_d} [\mathbb{E}_{x \sim p_{data}} [\log D(x)] + \mathbb{E}_{z \sim p_Z} [\log(1 - D(G(z)))] \quad (3)$$

The function code for creating the training data loader is shown below. The images are read directly from the folder with `torchvision.datasets.ImageFolder`. `tvv.Resize` can make sure that the images have the desired size. Then normalize the image pixel values to  $[-1, 1]$  with `tvv.ToTensor` and `tvv.Normalize`. Finally, return the `train_dataloader` instance for training. The batch size is 32, and the number of workers is 2.

```

1 | ...
2 | # create data loader
3 | def create_dataloader(args):
4 |     dataset = torchvision.datasets.ImageFolder(root=args.imgs_path,
5 |                                               transform = tvv.Compose([tvv.Resize(args.img_size),
6 |                                                                       tvv.ToTensor(),
7 |                                                                       tvv.Normalize((0.5, 0.5,
    |                                                                       ↪ 0.5), (0.5, 0.5, 0.5)
    |                                                                       ↪ ))

```

```

8 |                                     ]))
9 |     train_dataloader = DataLoader(dataset, batch_size=args.batch_size,
  |     ↪ shuffle=True, num_workers=args.n_workers)
10 |     return train_dataloader
11 | ...

```

The training function code is shown below. Both generator and discriminator use `torch.optim.Adam` as the optimizers with  $\beta_1 = 0.5$ ,  $\beta_2 = 0.99$ , and learning rate  $lr = 1e-4$ . The model weights are initialized with `weight_init` function. the number of epochs is 30. The number of channels of  $1 \times 1$  input noise vectors for the generator is 100. The label for a real image is 1, and that for a fake image is 0. The loss criterion is `nn.BCELoss`. For the discriminator, since it is the maximization step, the training target for real images is 1, according to the first item in Eq. 3; while the training target for fake images is 0, according to the second item in Eq. 3. For the generator, which is only contained in the second item of Eq. 3, the minimization of that item means the maximization of  $D(G(z))$ , which would encourage the generator to produce high-quality fake images. Besides periodically printing the losses, qualitative tracking by saving generated fake images is also implemented, which would be used to generate a `.gif` file. The `plot_learning_curve` plots the training losses versus iteration. In addition, a batch of real and fake images are saved for the demonstration purpose.

```

1 | ...
2 | # initialize weights
3 | def weight_init(m):
4 |     classname = m.__class__.__name__
5 |     # initialize (transposed) convolutional layers
6 |     if classname.find('Conv') != -1:
7 |         m.weight.data.normal_(0.0, 0.02)
8 |     # initialize batch normalization layers
9 |     elif classname.find('BatchNorm') != -1:
10 |         m.weight.data.normal_(1.0, 0.02)
11 |         m.bias.data.fill_(0)
12 |
13 | # plot the learning curves
14 | def plot_learning_curve(args, G_loss_list, D_loss_list):
15 |     # generate x-axis
16 |     x = [i for i in range(len(G_loss_list))]
17 |     plt.figure(figsize=(10, 5))
18 |     # plot each curve
19 |     plt.plot(x, G_loss_list, label='Generator')
20 |     plt.plot(x, D_loss_list, label='Discriminator')
21 |     plt.xlabel('iterations')
22 |     plt.ylabel('losses')
23 |     plt.legend()
24 |     cf = plt.gcf()
25 |     cf.savefig('{}loss.png'.format(args.result_path), bbox_inches='
  |     ↪ tight', format='png')
26 |
27 | # train the model
28 | def train(args, device):
29 |     # create a folder to save results
30 |     dir_name_for_results = args.result_path
31 |     if os.path.exists(dir_name_for_results):
32 |         files = glob.glob(dir_name_for_results + "/*")
33 |         for file in files:

```

```

34         if os.path.isfile(file):
35             os.remove(file)
36         else:
37             files = glob.glob(file + "/*")
38             list(map(lambda x: os.remove(x), files))
39     else:
40         os.mkdir(dir_name_for_results)
41     # create training data loader
42     train_dataloader = create_dataloader(args)
43     # fixed noise to track performance during training
44     noise_fixed = torch.randn(args.batch_size, args.nz, 1, 1, device=
45         ↪ device)
46     # create model instance
47     generator = GeneratorDCGAN().to(device)
48     discriminator = DiscriminatorDCGAN().to(device)
49     # iniitalize weights
50     generator.apply(weight_init)
51     discriminator.apply(weight_init)
52     # count number of learnable parameters
53     num_learnable_params_gen = sum(p.numel() for p in generator.
54         ↪ parameters() if p.requires_grad)
55     print('The number of learnable parameters in the Generator: %d'%
56         ↪ num_learnable_params_gen)
57     num_learnable_params_disc = sum(p.numel() for p in discriminator.
58         ↪ parameters() if p.requires_grad)
59     print('The number of learnable parameters in the Discriminator: %d'
60         ↪ %num_learnable_params_disc)
61     # count number of layers
62     num_layers_gen = len(list(generator.parameters()))
63     print('The number of layers in the generator: %d'%num_layers_gen)
64     num_layers_disc = len(list(discriminator.parameters()))
65     print('The number of layers in the discriminator: %d'%
66         ↪ num_layers_disc)
67     # use Adam optimizer for both networks
68     optimizer_G = torch.optim.Adam(generator.parameters(), lr=args.lr,
69         ↪ betas=(args.beta1,0.99))
70     optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=args.
71         ↪ lr, betas=(args.beta1,0.99))
72     # use BCE loss
73     criterion = nn.BCELoss()
74     # list to save losses
75     G_loss_list = []
76     D_loss_list = []
77     # labels for real and fake images
78     real_label_val = 1
79     fake_label_val = 0
80     # track iteration index
81     iters = 0
82     # list to save generated fake images during training
83     imgs_list = []
84     for epoch in range(args.n_epochs):
85         # list to save losses for printing
86         G_loss_list_temp = []
87         D_loss_list_temp = []

```

```

80     for i, data in enumerate(train_dataloader):
81         imgs_real = data[0].to(device)
82         # actual number of images in current batch
83         b_size = imgs_real.size(0)
84         # create labels for real and fake images
85         label_real = torch.full((b_size,), real_label_val, dtype=
            ↳ torch.float, device=device)
86         label_fake = torch.full((b_size,), fake_label_val, dtype=
            ↳ torch.float, device=device)
87         # train discriminator
88         discriminator.zero_grad()
89         output = discriminator(imgs_real).view(-1)
90         D_loss_real = criterion(output, label_real)
91         D_loss_real.backward() # gradient from real images
92         noise = torch.randn(b_size, args.nz, 1, 1, device=device)
93         fakes = generator(noise) # generator fake images
94         output = discriminator(fakes.detach()).view(-1)
95         D_loss_fake = criterion(output, label_fake)
96         D_loss_fake.backward() # gradient from fake images
97         optimizer_D.step() # update discriminator
98         D_loss = D_loss_real + D_loss_fake
99         D_loss_list_temp.append(D_loss)
100        # train generator
101        generator.zero_grad()
102        output = discriminator(fakes).view(-1)
103        G_loss = criterion(output, label_real)
104        G_loss.backward() # gradient from generated images
105        optimizer_G.step() # update generator
106        G_loss_list_temp.append(G_loss)
107        # periodically print losses
108        if i%args.print_every == args.print_every-1:
109            mean_G_loss = torch.mean(torch.FloatTensor(
                ↳ G_loss_list_temp))
110            mean_D_loss = torch.mean(torch.FloatTensor(
                ↳ D_loss_list_temp))
111            print('[epoch=%d/%d iter=%4d] mean_D_loss=%7.4f
                ↳ mean_G_loss=%7.4f]' % ((epoch+1), args.n_epochs, (i
                ↳ +1), mean_D_loss, mean_G_loss))
112            G_loss_list_temp = []
113            D_loss_list_temp = []
114        # save losses
115        G_loss_list.append(G_loss.item())
116        D_loss_list.append(D_loss.item())
117        # periodically visualize generated images
118        if (iters%args.generate_every == 0) or ((epoch == args.
            ↳ n_epochs-1) and (i == len(train_dataloader)-1)):
119            with torch.no_grad():
120                fake = generator(noise_fixed).detach().cpu()
121                imgs_list.append(torchvision.utils.make_grid(fake,
                    ↳ padding=1, pad_value=1, normalize=True))
122            iters += 1
123        # save the trained network
124        torch.save(generator, '{}generator.pth'.format(args.result_path))
125        torch.save(discriminator, '{}discriminator.pth'.format(args.

```

```

    ↪ result_path))
126 # plot learning curve
127 plot_learning_curve(args, G_loss_list, D_loss_list)
128 # generate GIF
129 imgs = []
130 for imgobj in imgs_list:
131     img = tvtf.to_pil_image(imgobj)
132     imgs.append(img)
133 imageio.mimsave('{}/generation_animation.gif'.format(args.
    ↪ result_path), imgs, fps=5)
134 # visualize real image samples from the dataset
135 real_batch = next(iter(train_dataloader))[0]
136 cv2.imwrite('{}/real_imgs.png'.format(args.result_path), np.flip(
    ↪ torchvision.utils.make_grid(real_batch.to(device), padding=1,
    ↪ pad_value=1, normalize=True).cpu().numpy().transpose(1,2,0)
    ↪ ,2)*255)
137 # visualize fake image samples
138 cv2.imwrite('{}/fake_imgs.png'.format(args.result_path), np.flip(
    ↪ imgs_list[-1].numpy().transpose(1,2,0),2).clip(0,1)*255)
139 ...

```

The adversarial losses over training iterations for both the generator and discriminator can be found in Fig. 2.

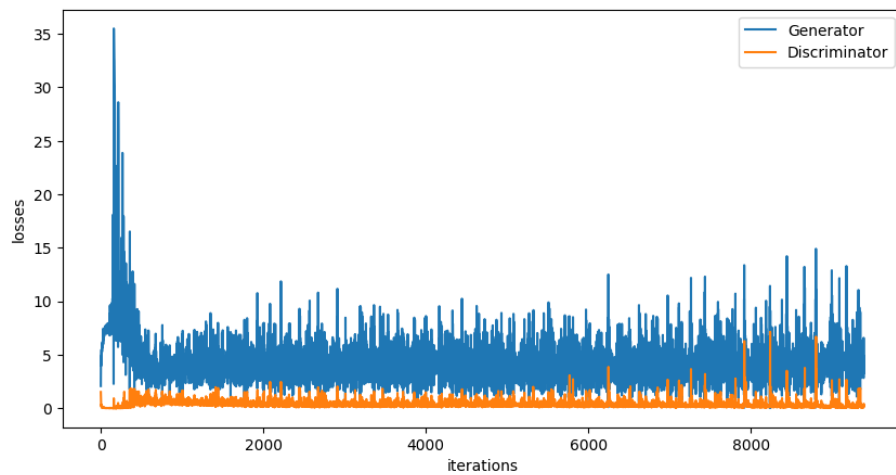


Figure 2: Adversarial losses over training iterations.

### 1.3 Comparison with Diffusion and evaluation

Note: The code for this subsection is borrowed from the DLStudio package (<https://engineering.purdue.edu/kak/distDLS/>).

#### 1.3.1 Qualitative evaluation

To qualitatively evaluate the performance of the GAN and Diffusion models, plot  $4 \times 4$  image grids. The code for plotting the real images and the fake images by GAN is shown below. In the `generate_img_demo` function, change the batch size `args.batch_size` to 16. And run the generator to generate 16 fake images.

```

1 ...
2 # generate samples
3 def generate_img_demo(args, device):
4     # change the batch size to 16 for the 4 by 4 demo
5     args.batch_size = 16
6     # create training data loader
7     train_dataloader = create_dataloader(args)
8     # load the trained network
9     generator = torch.load('{}generator.pth'.format(args.result_path)).
        ↳ to(device).eval()
10    # visualize real image samples from the dataset
11    real_batch = next(iter(train_dataloader))[0]
12    cv2.imwrite('{}real_imgs_demo.png'.format(args.result_path), np.
        ↳ flip(torchvision.utils.make_grid(real_batch.to(device), nrow
        ↳ =4, padding=1, pad_value=1, normalize=True).detach().cpu().
        ↳ numpy().transpose(1,2,0),2)*255)
13    # visualize fake image samples
14    noise = torch.randn(args.batch_size, args.nz, 1, 1, device=device)
15    with torch.no_grad():
16        fake = generator(noise).detach().cpu()
17    cv2.imwrite('{}fake_imgs_demo.png'.format(args.result_path), np.
        ↳ flip(torchvision.utils.make_grid(fake, nrow=4, padding=1,
        ↳ pad_value=1, normalize=True).numpy().transpose(1,2,0),2).clip
        ↳ (0,1)*255)
18 ...

```

The VisualizeSamples.py file in ExamplesDiffusion folder of DLStudio also needs to be modified to generate visualize  $4 \times 4$  image grids and save sample with the resolution of  $64 \times 64$ . The modified version VisualizeSamples\_modified.py is shown below. To be concise, only the lines that are modified or added are commented. For example, image\_display\_size is changed as (64,64); the code for saving  $4 \times 4$  image grid is rewritten using torchvision.utils.make\_grid.

```

1 ## VisualizeSamples_modified.py
2
3 ## Avi Kak, March 2024
4 ## modified by Wei Xu for HW8
5
6 import os,sys
7 import numpy as np
8 import torch
9 import torchvision
10 import matplotlib.pyplot as plt
11 import glob
12 from PIL import Image
13 import cv2 # import OpenCV package for saving image grid
14
15 print("\n\nPutting together a collage of the generated images for
        ↳ display\n")
16 print("\nFor the individual images, check the directory '
        ↳ visualize_samples'\n\n")
17
18 npz_archive = "RESULTS/samples_2048x64x64x3.npz"
19 visualization_dir = "visualize_samples"
20 image_display_size = (64,64) # changed the display size to 64 by 64
21

```



```

22 if os.path.exists(visualization_dir):
23     files = glob.glob(visualization_dir + "/*")
24     for file in files:
25         if os.path.isfile(file):
26             os.remove(file)
27 else:
28     os.mkdir(visualization_dir)
29
30 data = np.load(npz_archive)
31
32 for i, arr in enumerate(data['arr_0']):
33     img = Image.fromarray( arr )
34     img = img.resize( image_display_size )
35     img.save( f"visualize_samples/test_{i}.jpg" )
36
37 # generate image grid
38 im_tensor_all = torch.from_numpy(data['arr_0']).float()
39 im_tensor_all = torch.transpose(im_tensor_all, 1,3)
40 im_tensor_all = torch.transpose(im_tensor_all, 2,3)
41 cv2.imwrite(visualization_dir+"fake_images.png", np.flip(torchvision.
    ↪ utils.make_grid(im_tensor_all, nrow=4, padding=1, pad_value=1,
    ↪ normalize=True).cpu().numpy().transpose(1,2,0),2)*255)

```

Sample real images, sample fake images by GAN, and sample fake images by Diffusion (with 1000 diffusion time steps) can be found in Fig. 3, Fig. 4, and Fig. 5 respectively. Comparing the images from Fig. 4 and Fig. 5, the GAN produces more blurred parts, especially around the hair areas; the outputs of both GAN and Diffusion models are generally good, with the basic qualities of the human face; both models tend to output solid color backgrounds; both models output some noise with weird-value pixels.



Figure 3: Sample real images.

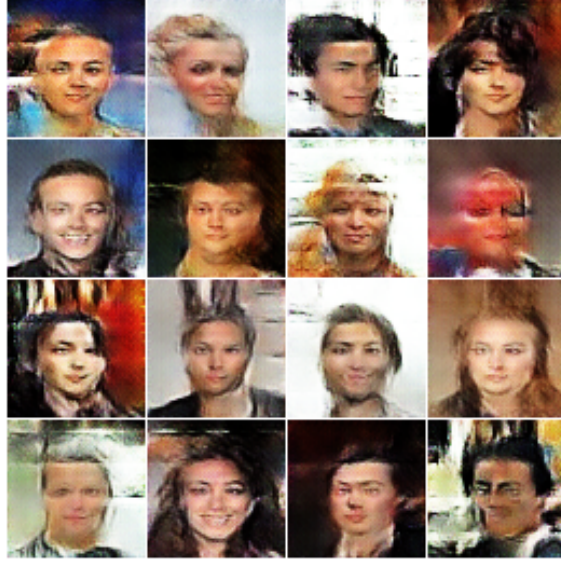


Figure 4: Sample fake images by GAN.



Figure 5: Sample fake images by Diffusion.

### 1.3.2 Quantitative evaluation

In this experiment, a quantitative evaluation using Frechet Inception Distance (FID) is performed. The code is shown below. 2048 fake images are generated in `generate_fake` function. Then the FID is calculated in `calculate_fid` function using `calculate_frechet_distance`.

```

1 | ...
2 | # generate fake images
3 | def generate_fake(args, device):
4 |     # create a folder to save images
5 |     dir_name_for_results = args.fake_path
6 |     if os.path.exists(dir_name_for_results):

```

```

7     files = glob.glob(dir_name_for_results + "/*")
8     for file in files:
9         if os.path.isfile(file):
10             os.remove(file)
11         else:
12             files = glob.glob(file + "/*")
13             list(map(lambda x: os.remove(x), files))
14     else:
15         os.mkdir(dir_name_for_results)
16     # load the trained network
17     generator = torch.load('{}generator.pth'.format(args.result_path)).
18         ↪ to(device).eval()
19     # create NumPy array to save image data
20     images = np.zeros((args.n_fake_sample, args.img_size[0], args.
21         ↪ img_size[0], 3), dtype=np.uint8)
22     # generate images one by one
23     with torch.no_grad():
24         for i in range(args.n_fake_sample):
25             noise = torch.randn(1, args.nz, 1, 1, device=device)
26             fake = generator(noise).detach().cpu()
27             images[i,:,:,:] = (np.flip(fake[0,:,:,:].detach().cpu().
28                 ↪ numpy().transpose(1,2,0),2).clip(-1,1)+1)*127.5
29             # save current image
30             cv2.imwrite('{}fake_imgs_{}.jpg'.format(args.fake_path,i),
31                 ↪ images[i,:,:,:])
32     # save NumPy array
33     np.save('{}imgs_fake_dcgan.npy'.format(args.result_path), images)
34
35 # calculate FID
36 def calculate_fid(args, device):
37     # retrieve image paths
38     real_path = sorted(glob.glob(os.path.join(args.imgs_path, '1/*.jpg'
39         ↪ )))[:args.n_fake_sample] # real images
40     # fake_path = sorted(glob.glob(os.path.join(args.fake_path, '*.jpg'
41         ↪ '))) # fake images by DCGAN
42     fake_path = sorted(glob.glob('../ExamplesDiffusion/
43         ↪ visualize_samples/*.jpg')) # fake images by Diffusion
44     # calculate activation statistics
45     dims = 2048
46     block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
47     model = InceptionV3([block_idx]).to(device)
48     m1, s1 = calculate_activation_statistics(real_path, model, device=
49         ↪ device)
50     m2, s2 = calculate_activation_statistics(fake_path, model, device=
51         ↪ device)
52     # calculate Frechet Inception Distance
53     fid_value = calculate_frechet_distance(m1, s1, m2, s2)
54     print(f'FID: {fid_value:.2f}')
55 ...

```

The FID for GAN and Diffusion models can be found in Tab. 1. The Diffusion model has a lower FID.

Table 1: FID for GAN and Diffusion models.

| Model     | FID   |
|-----------|-------|
| GAN       | 80.90 |
| Diffusion | 49.49 |

### 1.3.3 Discussion

According to the comparison, the Diffusion model is better in terms of image quality. First, the outputs from the Diffusion model have clearer and sharper boundaries, and they contain more details of the faces. Second, the FID of the diffusion model is lower.

However, the Diffusion is much slower. It takes several hours for Diffusion to generate 2048 fake images, while it only needs less than 10 seconds for GAN to generate the same amount of fake images.

To summarize, the trade-off between the image quality and running time needs to be considered. If the image quality is more important, the Diffusion model would be a better choice, but if the running time is more important, the GAN model could be a better choice.

## 2 Source Code

```

1 import os
2 import numpy as np
3 import random
4 import torch
5 import torch.nn as nn
6 from torch.utils.data import Dataset, DataLoader
7 import torchvision
8 import torchvision.transforms as tvt
9 import torchvision.transforms.functional as tvtF
10 import cv2
11 import imageio
12 import matplotlib.pyplot as plt
13 import glob
14 from pytorch_fid.fid_score import calculate_activation_statistics,
    ↪ calculate_frechet_distance
15 from pytorch_fid.inception import InceptionV3
16 import argparse
17
18 # set random seed
19 def set_random_seed(seed):
20     random.seed(seed)
21     torch.manual_seed(seed)
22     torch.cuda.manual_seed(seed)
23     np.random.seed(seed)
24     torch.backends.cudnn.deterministic = True
25     torch.backends.cudnn.benchmark = False
26     os.environ['PYTHONHASHSEED'] = str(seed)
27
28 # create data loader

```

```

29 def create_dataloader(args):
30     dataset = torchvision.datasets.ImageFolder(root=args.imgs_path,
31         transform = tvl.Compose([tvl.Resize(args.img_size),
32             tvl.ToTensor(),
33             tvl.Normalize((0.5, 0.5,
34                 ↪ 0.5), (0.5, 0.5, 0.5)
35                 ↪ )
36             ]))
37
38     train_dataloader = DataLoader(dataset, batch_size=args.batch_size,
39         ↪ shuffle=True, num_workers=args.n_workers)
40     return train_dataloader
41
42 # class for generator
43 class GeneratorDCGAN(nn.Module):
44     def __init__(self):
45         super().__init__()
46         # transposed convolutional layer with batch normalizaiton and
47         ↪ ReLU activation
48     def trans_conv_layer(in_channels, out_channels, kernel_size,
49         ↪ stride=1, padding=0):
50         return nn.Sequential(nn.ConvTranspose2d(in_channels,
51             ↪ out_channels, kernel_size=kernel_size, stride=stride,
52             ↪ padding=padding, bias=False),
53             nn.BatchNorm2d(out_channels),
54             nn.ReLU(inplace=True))
55
56     # define layers
57     self.latent2image = trans_conv_layer(100, 512, 4, 1, 0)
58     self.upsampler2 = trans_conv_layer(512, 256, 4, 2, 1)
59     self.upsampler3 = trans_conv_layer(256, 128, 4, 2, 1)
60     self.upsampler4 = trans_conv_layer(128, 64, 4, 2, 1)
61     self.upsampler5 = nn.ConvTranspose2d(64, 3, kernel_size=4,
62         ↪ stride=2, padding=1, bias=False)
63     self.tanh = nn.Tanh()
64     def forward(self, z):
65         z = self.latent2image(z)
66         z = self.upsampler2(z)
67         z = self.upsampler3(z)
68         z = self.upsampler4(z)
69         z = self.upsampler5(z)
70         z = self.tanh(z)
71         return z
72
73 # class for discriminator
74 class DiscriminatorDCGAN(nn.Module):
75     def __init__(self):
76         super().__init__()
77         # convolutional layer with batch normalizaiton (optional) and
78         ↪ ReLU activation
79     def conv_layer(in_channels, out_channels, kernel_size, stride
80         ↪ =1, padding=0, bn=True):
81         if bn:
82             return nn.Sequential(nn.Conv2d(in_channels,
83                 ↪ out_channels, kernel_size=kernel_size, stride=
84                 ↪ stride, padding=padding),

```

```

71         nn.BatchNorm2d(out_channels),
72         nn.LeakyReLU(negative_slope=0.2,
73                       ↪ inplace=True))
74     else:
75         return nn.Sequential(nn.Conv2d(in_channels,
76                                         ↪ out_channels, kernel_size=kernel_size, stride=
77                                         ↪ stride, padding=padding),
78                               nn.LeakyReLU(negative_slope=0.2,
79                                             ↪ inplace=True))
76     # define layers
77     self.conv1 = conv_layer(3, 64, 4, 2, 1, False) # first
78     ↪ convolutional layer does not have batch normalization
79     self.conv2 = conv_layer(64, 128, 4, 2, 1)
80     self.conv3 = conv_layer(128, 256, 4, 2, 1)
81     self.conv4 = conv_layer(256, 512, 4, 2, 1)
82     self.conv5 = nn.Conv2d(512, 1, 4, 1, 0)
83     self.sig = nn.Sigmoid()
84     def forward(self, x):
85         x = self.conv1(x)
86         x = self.conv2(x)
87         x = self.conv3(x)
88         x = self.conv4(x)
89         x = self.conv5(x)
90         x = self.sig(x)
91         return x
92 # initialize weights
93 def weight_init(m):
94     classname = m.__class__.__name__
95     # initialize (transposed) convolutional layers
96     if classname.find('Conv') != -1:
97         m.weight.data.normal_(0.0, 0.02)
98     # initialize batch normalization layers
99     elif classname.find('BatchNorm') != -1:
100         m.weight.data.normal_(1.0, 0.02)
101         m.bias.data.fill_(0)
102
103 # plot the learning curves
104 def plot_learning_curve(args, G_loss_list, D_loss_list):
105     # generate x-axis
106     x = [i for i in range(len(G_loss_list))]
107     plt.figure(figsize=(10, 5))
108     # plot each curve
109     plt.plot(x, G_loss_list, label='Generator')
110     plt.plot(x, D_loss_list, label='Discriminator')
111     plt.xlabel('iterations')
112     plt.ylabel('losses')
113     plt.legend()
114     cf = plt.gcf()
115     cf.savefig('{}loss.png'.format(args.result_path), bbox_inches='
116               ↪ tight', format='png')
117
118 # train the model
119 def train(args, device):

```

```

119 # create a folder to save results
120 dir_name_for_results = args.result_path
121 if os.path.exists(dir_name_for_results):
122     files = glob.glob(dir_name_for_results + "/*")
123     for file in files:
124         if os.path.isfile(file):
125             os.remove(file)
126         else:
127             files = glob.glob(file + "/*")
128             list(map(lambda x: os.remove(x), files))
129 else:
130     os.mkdir(dir_name_for_results)
131 # create training data loader
132 train_dataloader = create_dataloader(args)
133 # fixed noise to track performance during training
134 noise_fixed = torch.randn(args.batch_size, args.nz, 1, 1, device=
    ↪ device)
135 # create model instance
136 generator = GeneratorDCGAN().to(device)
137 discriminator = DiscriminatorDCGAN().to(device)
138 # initialize weights
139 generator.apply(weight_init)
140 discriminator.apply(weight_init)
141 # count number of learnable parameters
142 num_learnable_params_gen = sum(p.numel() for p in generator.
    ↪ parameters() if p.requires_grad)
143 print('The number of learnable parameters in the Generator: %d'%
    ↪ num_learnable_params_gen)
144 num_learnable_params_disc = sum(p.numel() for p in discriminator.
    ↪ parameters() if p.requires_grad)
145 print('The number of learnable parameters in the Discriminator: %d'%
    ↪ num_learnable_params_disc)
146 # count number of layers
147 num_layers_gen = len(list(generator.parameters()))
148 print('The number of layers in the generator: %d'%num_layers_gen)
149 num_layers_disc = len(list(discriminator.parameters()))
150 print('The number of layers in the discriminator: %d'%
    ↪ num_layers_disc)
151 # use Adam optimizer for both networks
152 optimizer_G = torch.optim.Adam(generator.parameters(), lr=args.lr,
    ↪ betas=(args.beta1, 0.99))
153 optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=args.
    ↪ lr, betas=(args.beta1, 0.99))
154 # use BCE loss
155 criterion = nn.BCELoss()
156 # list to save losses
157 G_loss_list = []
158 D_loss_list = []
159 # labels for real and fake images
160 real_label_val = 1
161 fake_label_val = 0
162 # track iteration index
163 iters = 0
164 # list to save generated fake images during training

```



```

165     imgs_list = []
166     for epoch in range(args.n_epochs):
167         # list to save losses for printing
168         G_loss_list_temp = []
169         D_loss_list_temp = []
170         for i, data in enumerate(train_dataloader):
171             imgs_real = data[0].to(device)
172             # actual number of images in current batch
173             b_size = imgs_real.size(0)
174             # create labels for real and fake images
175             label_real = torch.full((b_size,), real_label_val, dtype=
176                 ↪ torch.float, device=device)
177             label_fake = torch.full((b_size,), fake_label_val, dtype=
178                 ↪ torch.float, device=device)
179             # train discriminator
180             discriminator.zero_grad()
181             output = discriminator(imgs_real).view(-1)
182             D_loss_real = criterion(output, label_real)
183             D_loss_real.backward() # gradient from real images
184             noise = torch.randn(b_size, args.nz, 1, 1, device=device)
185             fakes = generator(noise) # generator fake images
186             output = discriminator(fakes.detach()).view(-1)
187             D_loss_fake = criterion(output, label_fake)
188             D_loss_fake.backward() # gradient from fake images
189             optimizer_D.step() # update discriminator
190             D_loss = D_loss_real + D_loss_fake
191             D_loss_list_temp.append(D_loss)
192             # train generator
193             generator.zero_grad()
194             output = discriminator(fakes).view(-1)
195             G_loss = criterion(output, label_real)
196             G_loss.backward() # gradient from generated images
197             optimizer_G.step() # update generator
198             G_loss_list_temp.append(G_loss)
199             # periodically print losses
200             if i%args.print_every == args.print_every-1:
201                 mean_G_loss = torch.mean(torch.FloatTensor(
202                     ↪ G_loss_list_temp))
203                 mean_D_loss = torch.mean(torch.FloatTensor(
204                     ↪ D_loss_list_temp))
205                 print('[epoch=%d/%d iter=%4d] mean_D_loss=%7.4f
206                     ↪ mean_G_loss=%7.4f' % ((epoch+1), args.n_epochs, (i
207                     ↪ +1), mean_D_loss, mean_G_loss))
208                 G_loss_list_temp = []
209                 D_loss_list_temp = []
210             # save losses
211             G_loss_list.append(G_loss.item())
212             D_loss_list.append(D_loss.item())
213             # periodically visualize generated images
214             if (iters%args.generate_every == 0) or ((epoch == args.
215                 ↪ n_epochs-1) and (i == len(train_dataloader)-1)):
216                 with torch.no_grad():
217                     fake = generator(noise_fixed).detach().cpu()
218                     imgs_list.append(torchvision.utils.make_grid(fake,

```



```

212         ↪ padding=1, pad_value=1, normalize=True))
213         iters += 1
214     # save the trained network
215     torch.save(generator, '{}generator.pth'.format(args.result_path))
216     torch.save(discriminator, '{}discriminator.pth'.format(args.
217         ↪ result_path))
218     # plot learning curve
219     plot_learning_curve(args, G_loss_list, D_loss_list)
220     # generate GIF
221     imgs = []
222     for imgobj in imgs_list:
223         img = tvtf.to_pil_image(imgobj)
224         imgs.append(img)
225     imageio.mimsave('{}generation_animation.gif'.format(args.
226         ↪ result_path), imgs, fps=5)
227     # visualize real image samples from the dataset
228     real_batch = next(iter(train_dataloader))[0]
229     cv2.imwrite('{}real_imgs.png'.format(args.result_path), np.flip(
230         ↪ torchvision.utils.make_grid(real_batch.to(device), padding=1,
231         ↪ pad_value=1, normalize=True).cpu().numpy().transpose(1,2,0)
232         ↪ ,2)*255)
233     # visualize fake image samples
234     cv2.imwrite('{}fake_imgs.png'.format(args.result_path), np.flip(
235         ↪ imgs_list[-1].numpy().transpose(1,2,0),2).clip(0,1)*255)
236
237     # generate samples
238     def generate_img_demo(args, device):
239         # change the batch size to 16 for the 4 by 4 demo
240         args.batch_size = 16
241         # create training data loader
242         train_dataloader = create_dataloader(args)
243         # load the trained network
244         generator = torch.load('{}generator.pth'.format(args.result_path)).
245             ↪ to(device).eval()
246         # visualize real image samples from the dataset
247         real_batch = next(iter(train_dataloader))[0]
248         cv2.imwrite('{}real_imgs_demo.png'.format(args.result_path), np.
249             ↪ flip(torchvision.utils.make_grid(real_batch.to(device), nrow
250             ↪ =4, padding=1, pad_value=1, normalize=True).detach().cpu().
251             ↪ numpy().transpose(1,2,0),2)*255)
252         # visualize fake image samples
253         noise = torch.randn(args.batch_size, args.nz, 1, 1, device=device)
254         with torch.no_grad():
255             fake = generator(noise).detach().cpu()
256         cv2.imwrite('{}fake_imgs_demo.png'.format(args.result_path), np.
257             ↪ flip(torchvision.utils.make_grid(fake, nrow=4, padding=1,
258             ↪ pad_value=1, normalize=True).numpy().transpose(1,2,0),2).clip
259             ↪ (0,1)*255)
260
261     # generate fake images
262     def generate_fake(args, device):
263         # create a folder to save images
264         dir_name_for_results = args.fake_path
265         if os.path.exists(dir_name_for_results):

```

```

252     files = glob.glob(dir_name_for_results + "/*")
253     for file in files:
254         if os.path.isfile(file):
255             os.remove(file)
256         else:
257             files = glob.glob(file + "/*")
258             list(map(lambda x: os.remove(x), files))
259     else:
260         os.mkdir(dir_name_for_results)
261     # load the trained network
262     generator = torch.load('{}generator.pth'.format(args.result_path)).
        ↳ to(device).eval()
263     # create NumPy array to save image data
264     images = np.zeros((args.n_fake_sample, args.img_size[0], args.
        ↳ img_size[0], 3), dtype=np.uint8)
265     # generate images one by one
266     with torch.no_grad():
267         for i in range(args.n_fake_sample):
268             noise = torch.randn(1, args.nz, 1, 1, device=device)
269             fake = generator(noise).detach().cpu()
270             images[i,:,:,:] = (np.flip(fake[0,:,:,:].detach().cpu().
        ↳ numpy().transpose(1,2,0),2).clip(-1,1)+1)*127.5
271             # save current image
272             cv2.imwrite('{}fake_imgs_{}.jpg'.format(args.fake_path,i),
        ↳ images[i,:,:,:])
273     # save NumPy array
274     np.save('{}imgs_fake_dcgan.npy'.format(args.result_path), images)
275
276     # calculate FID
277     def calculate_fid(args, device):
278         # retrieve image paths
279         real_path = sorted(glob.glob(os.path.join(args.imgs_path, '1/*.jpg'
        ↳ )))[:args.n_fake_sample] # real images
280         fake_path = sorted(glob.glob(os.path.join(args.fake_path, '*.jpg')))
        ↳ # fake images by DCGAN
281         # fake_path = sorted(glob.glob('../ExamplesDiffusion/
        ↳ visualize_samples/*.jpg')) # fake images by Diffusion
282         # calculate activation statistics
283         dims = 2048
284         block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
285         model = InceptionV3([block_idx]).to(device)
286         m1, s1 = calculate_activation_statistics(real_path, model, device=
        ↳ device)
287         m2, s2 = calculate_activation_statistics(fake_path, model, device=
        ↳ device)
288         # calculate Frechet Inception Distance
289         fid_value = calculate_frechet_distance(m1, s1, m2, s2)
290         print(f'FID: {fid_value:.2f}')
291
292     if __name__ == '__main__':
293         parser = argparse.ArgumentParser()
294         # '1' -- train the DCGAN model
295         # '2' -- generate 4 by 4 demos
296         # '3' -- generate 2048 fake images

```

```

297 # '4' -- calculate FID
298 parser.add_argument('--task', type=str, default='1',\
299     help='choose a task', choices=['1','2','3','4'])
300 parser.add_argument('--imgs_path', type=str, default='./celeba/',
301     ↪ help='dataset path')
302 parser.add_argument('--result_path', type=str, default='./
303     ↪ celeba_result/', help='dataset path')
304 parser.add_argument('--fake_path', type=str, default='./celeba_fake
305     ↪ /', help='fake path')
306 parser.add_argument('--print_every', type=int, default=100, help='
307     ↪ printing frequency')
308 parser.add_argument('--generate_every', type=int, default=500, help
309     ↪ ='generating frequency')
310 parser.add_argument('--n_epochs', type=int, default=30, help='batch
311     ↪ size')
312 parser.add_argument('--img_size', type=tuple, default=(64,64), help
313     ↪ ='image size')
314 parser.add_argument('--batch_size', type=int, default=32, help='
315     ↪ batch size')
316 parser.add_argument('--n_workers', type=int, default=2, help='
317     ↪ number of workers')
318 parser.add_argument('--nz', type=int, default=100, help='number of
319     ↪ channels of 1x1 input noise vectors for GeneratorDCGAN')
320 parser.add_argument('--lr', type=float, default=1e-4, help='
321     ↪ learning rate')
322 parser.add_argument('--beta1', type=float, default=0.5, help='
323     ↪ beta_1 for Adam optimizer')
324 parser.add_argument('--n_fake_sample', type=int, default=2048, help
325     ↪ ='number of fake images')
326 parser.add_argument('--cuda', type=str, default='cpu', help='Enable
327     ↪ cuda')
328 args = parser.parse_args()
329
330 device = torch.device(args.cuda)
331 seed = 0
332 set_random_seed(seed)
333
334 if args.task == '1':
335     train(args, device)
336 if args.task == '2':
337     generate_img_demo(args, device)
338 if args.task == '3':
339     generate_fake(args, device)
340 if args.task == '4':
341     calculate_fid(args, device)

```